
SKIN DEW
Your one-stop shop for dermatologist-recommended skincare—with the most accessible packaging on the market.
@dewySkin
@healthyskin

PROJECT DESIGN REPORT

SkinDEW – Online E-Store Web Application

VIDEO DEMO

---

1. Introduction

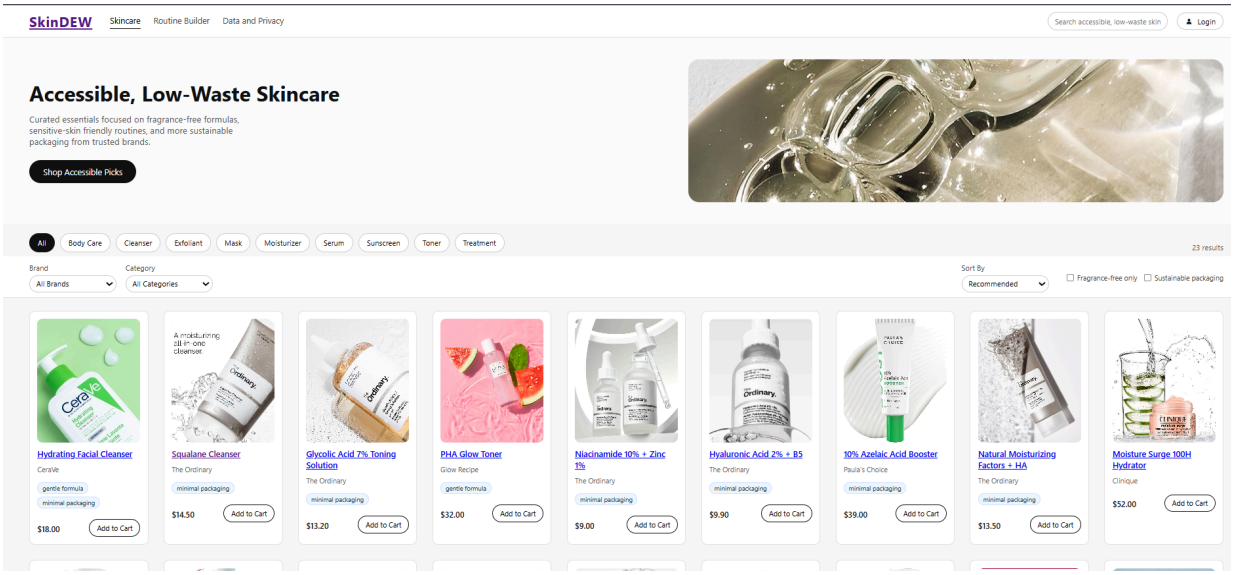
The SkinDEW project is an individual full-stack e-commerce web application designed to simulate the operations of a modern online skincare store. The platform serves as both a digital storefront and an educational hub that highlights eco-conscious brands using biodegradable, refillable, or minimal-waste packaging. Its goal is to make sustainable skincare products more accessible.

The system supports two primary user roles: customers, who browse products, manage carts, complete purchases, and maintain account information; and administrators, who oversee inventory, analyze sales data, and view or update customer records. The purpose of the project was to build a realistic online store using proper software engineering principles, including layered architecture, the MVC pattern, DAO abstraction, relational database design, and RESTful web APIs.

As the sole developer, I designed both the front-end and back-end from the ground up. On the backend, I implemented Node.js and Express to provide routing, middleware, controller logic, and a structured API layer. I used MySQL as the relational database for storing users, product inventory, shopping cart data, and purchase orders. On the front-end, I developed a responsive user interface using HTML, CSS, and vanilla JavaScript, integrating Fetch API calls to load

dynamic data. Throughout the development process, I applied design patterns such as MVC, DAO, and Service abstraction to maintain separation of concerns and ensure maintainability.

The main strengths of this design include its modularity, clear architectural separation, and complete support for major e-commerce workflows such as product browsing, sorting and filtering, cart management, checkout, and administrative reporting. A limitation is the use of static frontend pages instead of a reactive framework such as React, which could further improve state management and interactivity. However, given the scope, the chosen approach prioritizes clarity, simplicity, and direct alignment with course expectations.

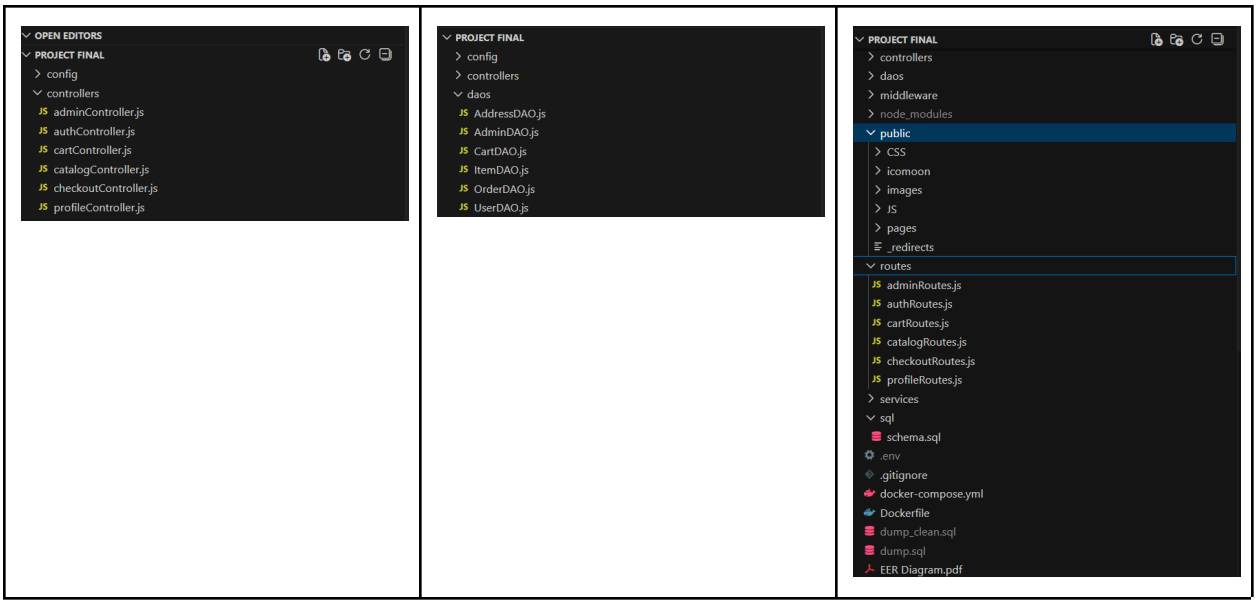


2. Architecture Description

The SkinDEW application follows a classical three-tier architecture consisting of a presentation layer, an application layer, and a data layer. This separation ensures that the user interface, business logic, and data storage remain independent and maintainable. The presentation layer contains all customer-facing and administrator-facing views, implemented as HTML/CSS/JavaScript files that run entirely in the browser. These pages interact with the backend strictly through REST APIs.

The application layer, built with Express.js, contains the controllers, middleware, and services that implement the main business logic. This includes authentication, catalog viewing, cart operations, checkout processing, user profile management, and administrative functions. Middleware enforces access control and validates user sessions, while the service layer handles specialized tasks such as payment simulation.

The data layer consists of a MySQL database, accessed exclusively through Data Access Objects (DAOs). Each DAO encapsulates all SQL operations for a specific domain entity, such as users, items, orders, or carts. This design prevents controllers from directly manipulating SQL queries and improves security, readability, and testability.



2.1 Architecture Overview

The main system components include:

- **Frontend Views:** Product catalogue, product details, shopping cart, checkout, profile management, registration/login, and admin dashboard.
- **Backend Controllers:** AuthController, CatalogController, CartController, CheckoutController, ProfileController, and AdminController.
- **Business Services:** PaymentService for simulated payment authorization.
- **Middleware:** Authentication middleware that enforces login and admin access rules.
- **Data Access Layer:** Six DAO modules (UserDAO, ItemDAO, CartDAO, AddressDAO, OrderDAO, AdminDAO).

- **Database:** MySQL schema with tables for users, items, carts, orders, order items, addresses, and payment methods.

All user actions follow a clear flow: the browser sends API requests → Express routes receive them → controllers execute logic → DAOs interact with the database → responses are returned to the frontend.

2.2 Use Case and Sequence Descriptions

Sequence Diagram:

1. Customer Browser Experience: This diagram shows the end-to-end flow of a typical shopping session. The customer's browser first requests the catalog from the server, which forwards the request to the `CatalogController` and `ItemDAO` to read products from the `item` table. When the customer views a single product or adds it to the cart, the browser sends requests to the `CartController`, which uses the session (user ID or guest session ID) and `CartDAO` to create or update entries in `cart_item`. During checkout, the `CheckoutController` reloads the cart from `cart_item`, verifies inventory against `item`, calls the `PaymentService`, and if successful, creates a `purchase_order` and related `purchase_order_item` records, updates stock, and clears the cart. The browser finally receives an order confirmation and displays it to the customer.

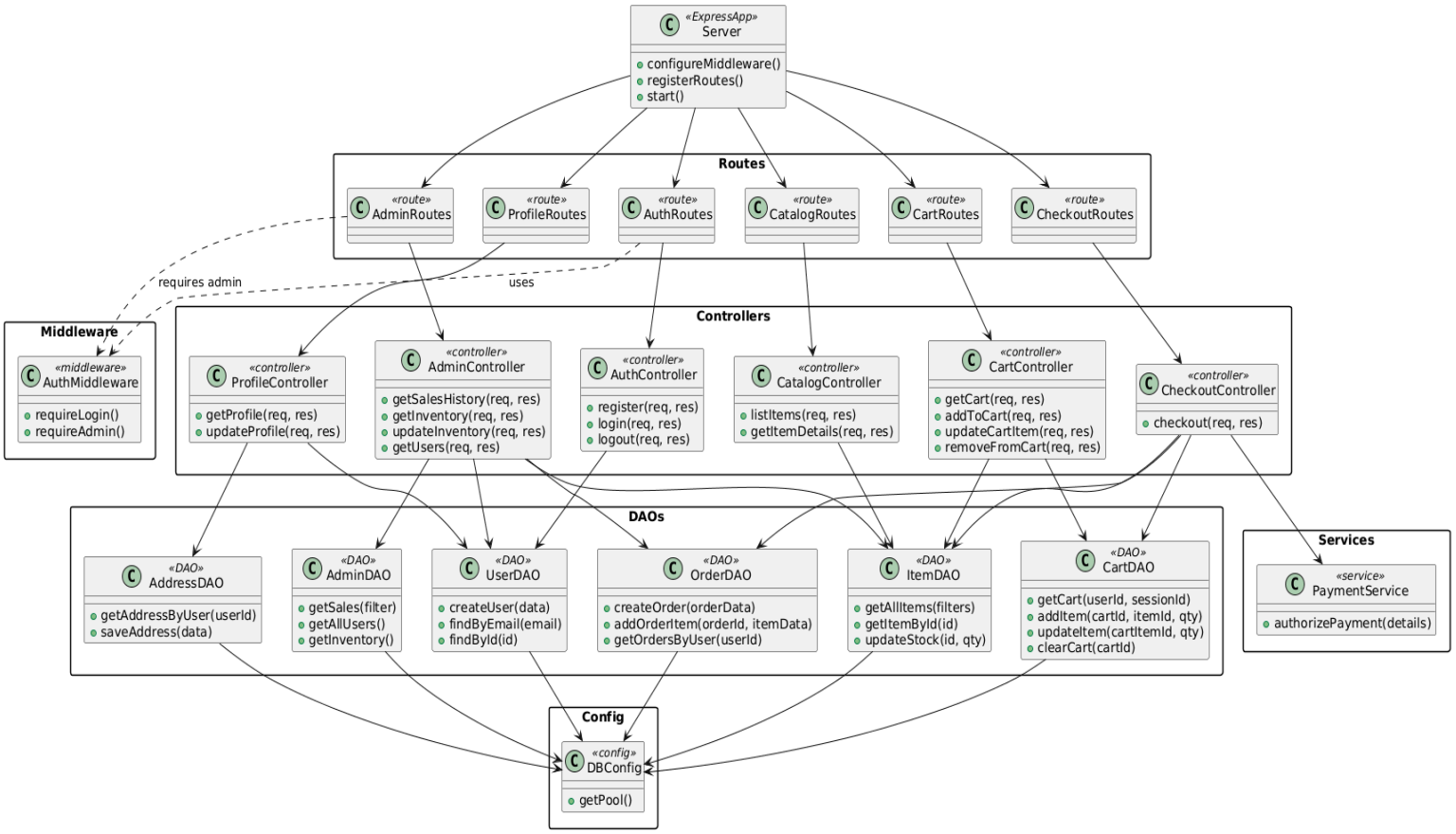
2. Admin Flow Diagram: This diagram illustrates how an administrator logs in and manages the store. The admin logs in through the browser, which sends credentials to the `AdminController`. The controller uses `UserDAO` to verify the user in the `user` table and checks the `is_admin` flag before storing admin details in the session. For dashboard statistics and sales history, the browser calls admin endpoints that route to `AdminController` and `AdminDAO`, which run aggregate and joined queries over `purchase_order`, `purchase_order_item`, `item`, and `user`. For inventory updates, the admin submits new quantities from the dashboard; the request goes to `AdminController`, which calls `AdminDAO` to update the `item` table. Successful updates are returned to the browser and reflected immediately in the admin UI.

3. Design Description

Our system extensively applies design patterns. The **MVC architecture** separates controllers (business logic), DAOs (data access), and views (presentation). The **DAO pattern** ensures database queries remain isolated and reusable, improving maintainability and reducing duplication. **Middleware design** implements cross-cutting concerns such as session validation and access control. The **Service pattern** is used for the payment module, which abstracts the logic behind payment authorization and could be swapped for a real provider using the same interface.

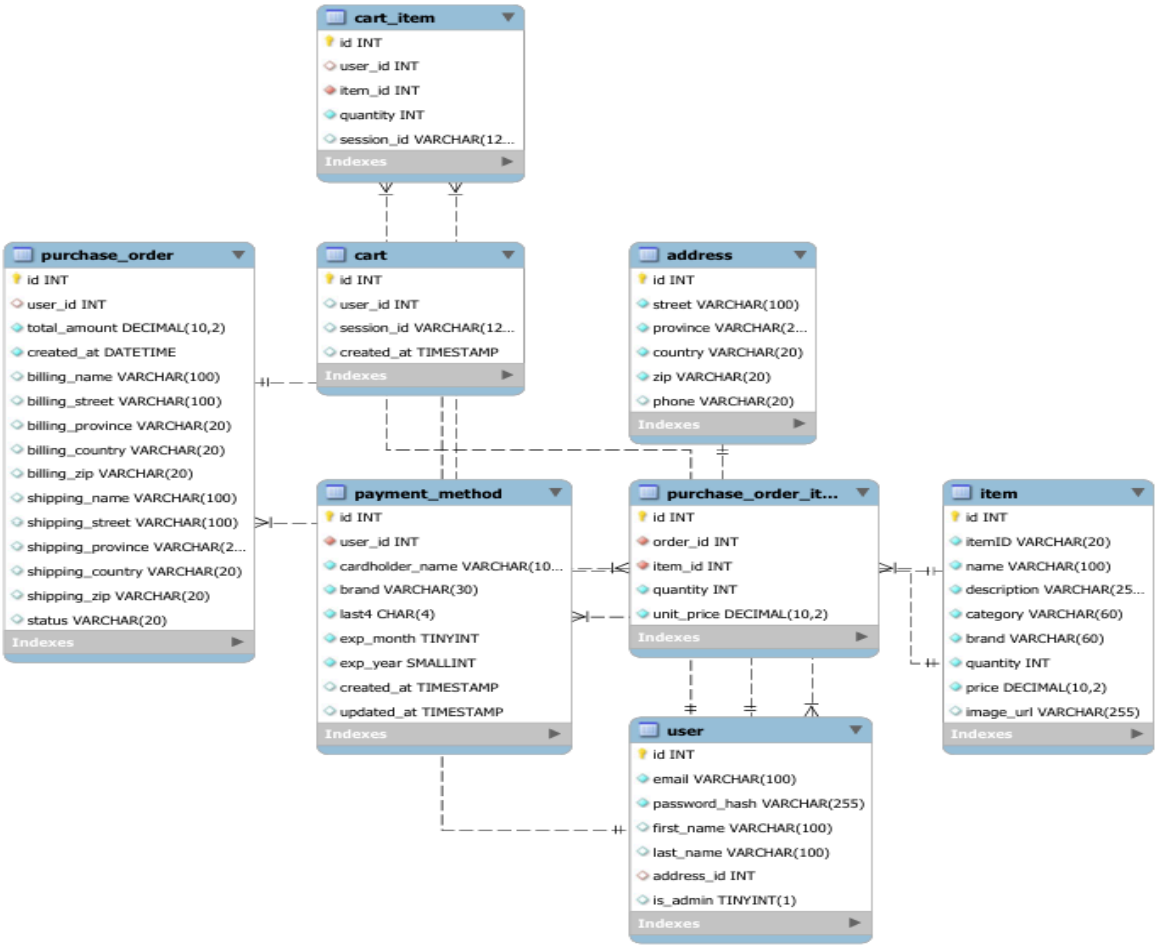
Trade-offs included selecting simplicity and clarity over advanced abstraction layers, ensuring the project remained understandable and met assignment requirements.

3.1 Class Diagram (Architecture Level)



The class diagram shows a layered architecture separating the web application into presentation, business logic, data access, and database domain classes. At the top, the `ExpressApp` class uses route classes (e.g., `AuthRoutes`, `CatalogRoutes`, `CartRoutes`, `CheckoutRoutes`, `ProfileRoutes`, `AdminRoutes`) to map HTTP endpoints to controller classes such as `AuthController`, `CatalogController`, `CartController`, `CheckoutController`, `ProfileController`, and `AdminController`.

Each controller implements the main use cases (authentication, catalog browsing, cart operations, checkout, profile management, and admin functions) and delegates all database



The database schema is implemented in MySQL and models the core entities of the SkinDEW e-commerce domain. The `user` table stores login credentials, personal details, and an `address_id` foreign key linking to the `address` table, which contains street, province, country, postal code, and phone information. The `payment_method` table is related to `user` via `user_id` and stores a masked representation of the customer's card (brand, last four digits, and expiry), allowing one user to have multiple saved payment options.

Products are represented by the `item` table, which includes fields such as `name`, `description`, `category`, `brand`, `price`, `quantity`, and `image_url`, enabling inventory tracking and image display on the frontend. Shopping activity is captured in the `cart_item` table, which associates each item with either a `user_id` or a `session_id` so that both logged-in and guest carts are supported. Completed purchases are stored in `purchase_order` and `purchase_order_item`: `purchase_order` records the overall order, including user, amounts, status, and denormalized billing and shipping details, while `purchase_order_item` links each order to one or more items with quantity and unit price. Together, these tables enforce referential integrity for orders and enable administrators to query sales, inventory levels, and customer histories.

4. Advanced Features

Guest Cart (Session-Based) Support: Cart works **both for logged-in users and guests** using `sessionID`, with items retained when navigating back to continue shopping.
The specification only required "can do this with or without login"; you implemented a full technical solution with sessions.

Password Hashing and Auth Middleware: Secure password storage (hashing) and middleware to protect profile/admin routes instead of just plain-text checks.

Admin Dashboard with Aggregated KPIs: Admins don't just see raw sales rows; they get **summary statistics** (total sales, order counts, etc.).

Enhanced Search & Filtering Behaviour: Search works across multiple fields (e.g., product name, description, category, brand) instead of a very basic single-field search. Combined use of **filter + search + sort** together on the catalogue.

Docker & Railway Deployment Setup: App is fully containerized with `Dockerfile` and `docker-compose`, and deployed on Railway with MySQL.

---

5. Implementation Section

During development, I evaluated several frontend and backend technologies before selecting the final stack. For the backend, I considered Java Servlets/JSP and Spring Boot, which were covered or discussed in the course. While both are powerful frameworks, Node.js and Express offer a faster development cycle, more flexible routing, and easier integration with JSON-based APIs. Express allows controllers, middleware, and services to be implemented in a modular way, closely aligning with the MVC structure required by the project.

On the frontend, I evaluated React and Angular as potential frameworks. Although both frameworks offer component-based rendering and state management, I ultimately chose to use standard HTML, CSS, and JavaScript because it allowed for a simpler and more transparent implementation of the user interface. With Fetch API, dynamic content loading could be handled effectively without the additional complexity of a reactive framework. This decision also reduced the learning curve and better fit the requirements of an academic project.

For the database layer, I compared MongoDB and MySQL. I chose MySQL because the project relies heavily on relational data, such as user-to-order relationships, order-to-item relationships, and cart structures that map naturally to foreign keys. MySQL also integrates smoothly with Railway, simplifying the deployment process.

---

6. Deployment Efforts

I attempted multiple deployment strategies, including local Docker builds and cloud-based approaches. Ultimately, Railway was chosen because it supports Node.js natively and provides a hosted MySQL instance. The deployment required creating environment variables, configuring the database pool, and exposing a healthcheck route. After resolving issues related to CORS, sessions, and route paths, the application was successfully deployed.

---

7. Conclusion

Completing the SkinDEW e-commerce application allowed me to apply multiple architectural patterns and software engineering concepts, including MVC structure, DAO abstraction, RESTful API design, relational database modeling, and full-stack implementation. The project demonstrates a complete, functioning online store with customer and administrator workflows, all supported by a modular and secure architecture.

What went well was the successful end-to-end integration of all layers (frontend, API, DAOs, and database) and a working cloud deployment; what went wrong initially were deployment misconfigurations and debugging asynchronous backend behaviour, which ultimately became valuable learning experiences. The main weaknesses stem from the use of static frontend technologies, which, while sufficient for the project, limit scalability and dynamic UI capabilities compared to frameworks such as React.

Working independently posed unique challenges, such as ensuring code quality without peer review and handling all architectural decisions alone. However, these challenges also contributed to a deeper understanding of the full development cycle. I learned how to design APIs, structure DAOs, manage session-based authentication, debug asynchronous backend behavior, and deploy real web applications using cloud tools. Completing the project alone reinforced problem-solving skills and provided experience that closely mirrors real-world full-stack development.